# Cache 與矩陣乘法

李哲榮

還記得程設二的時候教過的電腦系統嗎? 所有要計算的程式和資料都要先由硬碟讀到記憶體，然後程式執行的時候要將記憶體的資料讀到 CPU 的暫存器 registers 才可以執行。

但是 CPU 的速度和記憶體的速度實在差太多了，假設執行 z=x+y 的 CPU 時間是 1，把 x 和 y 由記憶體讀進 CPU 來的時間至少是 20-100，把 z 搬走的時間也差不多。也就是說 CPU 要執行一行指令要等 100 倍的時間，這是無法忍受的事，所以有了 cache 記憶體的發明。

Cache 記憶體是在 CPU 內部的記憶體，通常可以和 CPU/register 有類似的速度，如果可以把要計算的資料拿到 Cache 內，那 CPU 就不用一直發呆。唯一的缺點就是 cache 通常很小，而且如果要用的資料只用一次，把資料由 RAM 搬進去 cache 的時間和把資料直接讀到 CPU 的時間差不多，有了 cache 反而要多讀一次(cache to CPU)，所以 cache 的真正使用方法是，當 CPU 要資料時，資料已經在 cache 中了。

要達到這一點有兩個方法，第一就是 pre-fetch，CPU 知道接下來要用甚麼資料，所以先把要用的資料都先搬進來，當然"預測"這件事很難，但是所幸大部分的程式使用資料時都有 locality，也就是目前使用資料 M[a1]，接下來大概就會使用資料 M[a1+1]，M[a1+2]等等，所以只要把目前用的資料附近的資料一起搬進 cache，大概都可以預測得不錯。例如矩陣在 C 裡面是以 row major 的形式儲存，所以讀取時會把同一個 row，相鄰的數字也讀進 cache。但是光這樣是不夠的，因為 cache 還是太小，就算知道接下來的讀取那些資料，也是要一直讀取。

第二個有用的方法是將 cache 裡的資料反覆使用。這裡一個指標是 computation - communication ratio (CCR)，就是計算量和資料量的比值，簡單的說就是一個資料被搬進 cache 之後，會被使用幾次，如果 CCR 越高，搬資料相對的時間代價越低。我們以這次的矩陣乘法為例，比較兩種程式寫法的 CCR。

第一種寫法：

```
For i = 0: 1023
    For j = 0:1023
        For k=0:1023
            C[i,j] += A[i,k]*B[k,j];
```

每計算一個 C[i,j]，我們要把 A[i, 0:1023]和 B[0:1023, j]讀進來，假設 cache 可以儲存 3072 個數字，而且有完美的 pre-fetch，我們剛好可以放 C 的一個 row，A 的一個 row，和 B 的一個 column。我們一次 cache 的資料最多就是計算一個 C[i,j]。例如要計算 C[0,0]，我們需要把 A[0, 0:1023]和 B[0:1023, 0]放入 cache，計算量只有 2*1024，CCR 就是 4*1024/3072=2/3。之後要計算 C[0,1]，我們就要把 cache 中的 B[0:1023,0]換成 B[0:1023,1]，以此類推，所以我們要計算 C 的第一個 row，C[0,0:1023]，我們要把 B 整個 matrix 讀一遍到 cache，等到要計算下一個 row 的時候，又要在把 B 整個 matrix 讀一遍到 cache 一遍。計算完整個 C matrix，A 只要讀一遍，整個 B matrix 要讀 1024 遍。 $O(N^3)$ 的計算量需要 $O(N^3)$ 的資料搬移量，CCR 約等於 1。

第二種 block 的寫法，假設 block size b = 32，我們用 $A_b$, $B_b$, 和 $C_b$ 表示 block submatrices。Cache 中剛好可以放入一個 C 的 block, 一個 A 的 block，和一個 B 的 block。例如計算 $C_b[0,0]$時，我們要計算

$$C_b[0,0] = \sum_{k=0}^{31} A_b[0,k]B_b[k,0]$$

每一個$A_b[0,k]$和$B_b[k,0]$都可以被放到 cache 中，但是計算量是 2*32*32*32 = 64*1024，CCR 等於 64*1024/3072 =64/3，是第一種寫法的 32 倍。整體而言，如果我們要算完整個 C 矩陣，我們要把 A 讀 32 遍，把 B 也讀 32 遍，$O(N^3)$ 的計算量需要 $O(N^3/b)$ 的資料搬移量，CCR 約等於 b=32。

假設資料搬移時間遠大於計算時間，所以我們只考慮資料搬移時間。第一種方法要把 A 矩陣搬 1 遍，B 矩陣搬 1024 遍，C 矩陣搬 1 遍。第二種方法把 A 矩陣搬 32 遍，B 矩陣搬 32 遍，C 矩陣搬 1 遍。兩者的時間比是 1025/65 ~ 16 倍。

真實的狀況會更複雜，因為 cache 有不同架構，也有不同 replacement policy，也有 L1, L2, L3 等階層，這些等到你們修 architecture 時可以去了解。另外 compiler 也有影響，因為 compiler 會交換程式碼的順序，會 pre-fetch data，可以提高 CCR，還有許多神奇的效能最佳化方法，你們可以去修 compiler 來了解。OS 也有關聯，因為在執行程式時會有 multi-tasking，會 schedule 其他的程序來執行，你也可以透過 OS 設定程式執行的優先次序，綁定特定 core，另外 OS 也影響記憶體和虛擬記憶體的配置，這一部分影響效能更大，修了 OS 後會了解。但是，影響這兩個程式效能差異的主要原因仍然是他們的 CCR 不同。